Data Cube Computational Model with Hadoop MapReduce

Bo Wang[1], Hao Gui[1], Mark Roantree[2], Martin F. O'Connor[2]
[1] *International School of Software, Wuhan University, China*
[2] *Insight: Centre for Data Analytics, Dublin City University, Ireland*
{*bobwang,hgui*}@*whu.edu.cn*, {*mark.roantree,moconnor*}@*computing.dcu.ie*

Keywords: XML, Hadoop, Data Warehouse, MapReduce.

Abstract: XML has become a widely used and well structured data format for digital document handling and message transmission. To find useful knowledge in XML data, data warehouse and OLAP applications aimed at providing supports for decision making should be developed. Apache Hadoop is an open source cloud computing framework that provides a distributed file system for large scale data processing. In this paper, we discuss an XML data cube model which offers us the complete views to observe XML data, and present a basic algorithm to implement its building process on Hadoop. To improve the efficiency, an optimized algorithm more suitable for this kind of XML data is also proposed. The experimental results given in the paper prove the effectiveness of our optimization strategies.

1 INTRODUCTION

XML is a widely used and well structured data format especially in web-based information systems. Its flexible nature makes it possible to represent many kinds of data. The Web constantly offers new services and generates large volumes of new data in XML. However, it is difficult to satisfy the additional processing requirements necessary to facilitate OLAP applications or data mining with XML data. The underlying differences between the relational and XML data models present many challenges. It is difficult to provide a logical and direct mapping from one data model to the other due to the impedance mismatch between them (Rusu et al., 2009). In our previous works (Gui and Roantree, 2012) (Gui and Roantree, 2013), we have proposed a pipeline design based on an OLAP data cube construction framework designed for real time web generated sensor data. We transformed sensor data into an XML stream conforming to the data warehouse logical model and built a corresponding data cube tree and serialized it into an XML data cube representation.

In this paper, our research focuses on how to process large scale XML data efficiently. The concept of "cloud computing" has received considerable attention recently because it facilitates a solution to the increasing data demands through a shared and distributed computing infrastructure (Dutta et al., 2011). Apache Hadoop provides a powerful tool for tackling large-scale data problems in the area of machine learning, text processing, bioinformatics, etc. Hadoop implements a computational paradigm called MapReduce. The application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. In addition, it provides a distributed file system to store data and permits a very high throughput for aggregate operations across the node cluster. MapReduce has emerged as an attractive alternative: its functional abstraction provides an easy-to-understand model for designing scalable and distributed algorithms (Lin and Schatz, 2010). Recently there has been some research into the provision of a parallel processing computational model for XML documents over distributed systems (Dede et al., 2011). In (Khatchadourian et al., 2011), the authors present a language called ChuQL to express XML oriented data processing tasks on the cloud. XML is a semi-structured data format, and due to its distributed paradigm, Hadoop is well positioned to provide a reliable and scalable platform for processing semi-structured data. By transforming large-scale XML data into the data cube presented in this paper, it will be easier to process the data on Hadoop and significantly reduce the risk of data loss.

2 BACKGROUND

2.1 CityBike Project

The deployment of sensors in the physical world is constantly increasing and may now be regarded as widespread. The number of applications relying on sensor data is also growing. Examples include: urban traffic watch, weather monitoring, tracking of goods, etc. We now describe a project that serves as a use case for our work. The city of Dublin (along with many other European cities) has deployed a bike sharing scheme whereby people may rent (and return) bikes from stations located throughout the city center. The stations are equipped with sensors to monitor the availability of bikes, and the stations publish this information to the DublinBikes website (www.dublinbikes.ie). Consumers can connect to the website (either through a desktop PC or mobile application) to view this information, including the location of stations, the number of bikes currently available for renting, the number of spaces available to return bikes, and so on. For both consumers and providers of the service, the data is of great interest. Consumers can check where to rent or return a bike while providers can understand at which station it is best to pick up or return bikes for maintenance in order to minimize service disruption.

The bicycle rental application, available in many cities and towns across Europe, collects data at regular intervals from each location. For each station this information is made available through the service provider's web site which provides the station ID, the total number of bike stands, the number of bikes available, the number of free bike stands available, and so on. The bicycle rental statistics may also be connected with other related factors, such as weather conditions, bus routes in the city, and the location of local train stations.

2.2 XML data cube construction

The Data Warehouse (DW) definition is different from the schema of the underlying native XML database. The DW definition includes a high level description of all the dimensions and the fact data and the relationships between them (as shown in Fig.1). In contrast, the schema of the XML database focuses on the low-level logical and physical structure of the XML data. The initial web generated raw sensor data must be transformed and mapped to the structure described by our DW definition illustrated in Fig.1 It is not necessary to permanently materialize the identifier information in order to build up the XML DW

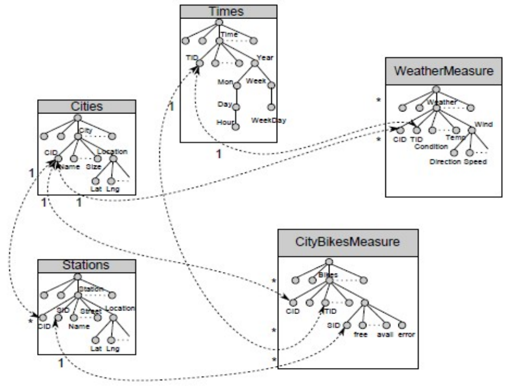


Figure 1: An example of logical model for the data warehouse in CityBikes Project (Gui and Roantree, 2013)

galaxy model and to specify the relationships between the different parts. (Gui and Roantree, 2013) The cube definition details the key components of the cube construction process: selected dimensions, selected concept hierarchy within dimensions, measurements of fact data and aggregating functions. This information can be used to determine both the construction process and the production of the data cube. In our project, we have developed several XML Schemata to model the data cube structure, which we name the XML Data Cube Model (XDCM). The data cube can be serialized into XML data and an associated index may be generated to facilitate OLAP operations on the data cube. Both the serialized XML data cube and the corresponding index (not discussed in this paper) may be stored into the underlying native XML database and be made available to related data analysis applications.

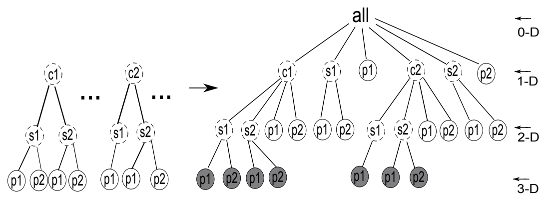


Figure 2: XDCM cube tree construction demonstration

In Fig.2, the forest on the left side is a simple illustration of the input XML data concerning bikes usage, which contains source information to be aggregated. To simplify the example, we only selected three dimensions and each dimension has no more than two different values. In the input forest, every node at each level may be repeated an arbitrary number of times. In the CityBike project, the system generates the data regularly, and in some dimensions, the domains are quite limited regarding the number of stations and the weather details. This will result in a large quantity of mergers in subsequent operations. The height of input tree is equivalent to the dimension in the DW. Assuming the dimension of input data is D, then we will get 2^D views to observe the data, including dimension 0 . Our XDCM must be robust enough to describe all the views directly without any join or related operations. In the case of the forest on the left side in Fig.2, using the capital letters to represent the dimensions, we could get $C-S-P$, $C-P$, $C-S$, $S-P$, C, S, P and Φ. In fact, $C-S-P$ includes $C-S$ and C, $S-P$ includes S. From a tree's point of view, you must visit the nodes from top to bottom along the paths. If you have reached P, it follows that you have already reached C and S. To reduce the redundancy, some views that are included by others will not be generated directly in the XDCM. In other words, all the branches of XDCM cube tree should end with the nodes in P in this demonstration. The Φ view is the "all" in the XDCM cube tree at level 0.

The tree on the right side is the XDCM tree be be constructed. There are three kinds of nodes that will be used to finalise the construction.

- **Data Node.** The gray nodes in Fig.2 represent the Data Nodes. After several processes such as filtration or combination (if required), the initial data can be generated on the Data Nodes directly. The Data Nodes contain basic information such as paths and values, and they describe the view with the longest path such as $C-S-P$ highlighted in the previous example.
- **Additional Node.** The Additional Node (indicated by a dashed-circle in Fig.2) which is produced by the Data Node is a form of intermediate data not shown in Fig.2. Each Data Node will produce all possible Additional Nodes through the removal of any constraints in the different dimensions in its path.
- **Link Node.** The white nodes represent the Link Nodes which describe other additions paths of observation. Link Nodes are produced by combining Additional Nodes that share the same paths. Nodes in different paths should be unique in the XDCM.

During the cube construction, for example, Data Node $all-c1-s1-p1$ will produce Additional Nodes $all-c1-p1$, $all-s1-p1$, $all-p1$, and Data Node $all-c1-s2-p1$ will produce Additional Nodes $all-c1-p1$, $all-s2-p1$, $all-p1$. Thus, there are two pairs of nodes that share the same paths. They are $all-s1-p1$ and $all-p1$. Upon combination of these nodes, we obtain Link Nodes $all-c1-p1$, $all-s1-p1$, $all-s2-p1$ and $all-p1$. Both Data Nodes and Link Nodes are the nodes in P. Other nodes describe other views can be calculated in following processes.

2.3 Map-Reduce

MapReduce is a programming framework for processing large data sets with a parallel, distributed algorithm on clusters. The most important functions in this model are Map and Reduce. Mappers receive a collection of Key-Value pairs and produce zero or more output Key-Value pairs. Pairs sharing the same key are collected and delivered to the same Reducer. Reducers can iterate through the values that are associated with that key and produce zero or more outputs. Hadoop has implemented this programming model and offers a well-defined set of APIs to control the entire process.

In Hadoop, a complete Map-Reduce task can be regarded as a job. To finish an entire task, it may be necessary to perform more than one job. Input data is split into many parts and transformed into Key-Value pairs. In this context, the concept of *split* is about a logical rather than a physical operation. Consequently, the data uploaded to the Hadoop Distributed File System (HDFS) will be cut into several blocks, usually in 64MB chunks, according to the requirements of the system. Hadoop is quite adept at dealing with unstructured data, such as text, because it is easy to split and suitable for the physical structure. Hadoop does not offer any functionality to process XML documents directly. Mahout, an open-source program based on Hadoop, provides a class to handle the XML input format, and the underlying strategy is simple. An XML document is well organized, the content of every element is enclosed by a pair of tags. Therefore, it is sufficient to simply read the XML document as a text file and select the content between the certain pair of tags.

3 CUBE CONSTRUCTION WITH MAP-REDUCE

In order to describe the algorithm, consider an exercise to build an OLAP data cube for the bicycle rental scenario under certain weather conditions and in various locations and dates. In this case, only three dimensions have been chosen as shown in Fig.2. The Input data are nodes in the dimension named P with some additional information, such as their path constraints. On the left side, the nodes indicated with a dashed-circle illustrate the paths of the nodes circled by solid lines, and they do not need to be instantiated. Our goal is to build the tree on the right side and to calculate the values of the dashed nodes.

For our implementation, the entire task can be divided into two phases. Phase one is to generate the Data Nodes and Link Nodes, and then organize them into the XDCM as described earlier. The second phase is to calculate the values of the dashed nodes on the right side which represent observation views other than Data Nodes and Link Nodes. The calculation task itself can be divided into two further subtasks. One subtask is used for the calculation of Data Nodes and the other subtask is used for the calculation of Link Nodes.

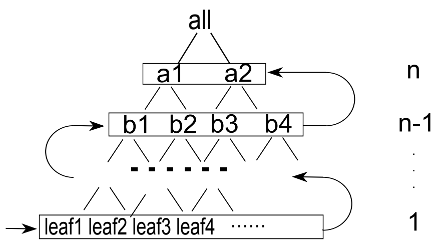


Figure 3: Processing Order of Data Nodes

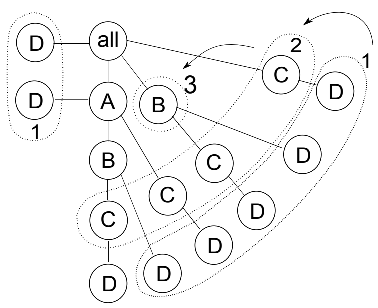


Figure 4: Processing Order of Link Nodes

Fig.3 illustrates the node processing order of the algorithm. The integer values on the right side represent the round indexes of Map-Reduce jobs. Data Nodes are the leaf nodes shown at the bottom, and they will be processed by the first job to generate Additional Nodes. The Additional Nodes will subsequently be merged into Link Nodes. Thus, the subsequent rounds only concentrate on the calculation tasks. For a N-dimensional XML data, we need at least N rounds to complete the entire construction, in other words, N jobs are needed to complete the whole task in Hadoop. The output generated from each job will be used as the input data for the next job. The computation receives a set of Key-Value pairs as input, and then produces a set of output Key-Value pairs:

$$Map \quad (k1, v1) \quad \rightarrow \quad list(k2, v2)$$
$$Reduce \quad (k2, list(v2)) \quad \rightarrow \quad (k3, v3)$$

In the generation phase (phase one), the initial data is transformed into Key-Value pairs that actually represent Data Nodes. So $k1$ is the path of a given Data Node, and $v1$ is the value of that node. Data Nodes will produce large amounts of Additional Nodes. $k2$ is the path of an Additional Node, and $v2$ is its value. The Reduce function accepts the intermediate key $k2$ and a set of values for that key, and merges these values to form a Link Node. So $k3$ is the path of Link Node, and $v3$ is the actual value.

In the calculation phase (phase two), the process starts with either the Data Nodes or the Link Nodes. All nodes in the same level will be processed in one job. The entire process proceeds in a bottom-up manner from one layer to the next. However, unlike the generation phase, the Key value of the Mapper's output is the path of the node without self-containment. For example, given the input node γ and the path $\alpha - \beta - \gamma$. Then $k2$ will be $\alpha - \beta$, and $v2$ will be γ. The Reduce function accepts the intermediate key $\alpha - \beta$ and calculates the value of β. The $k3$ is the complete path of β which is $\alpha - \beta$, and $v3$ is β. This $(k3, v3)$ pair will be the $(k1, v1)$ in the next calculation round, which is going to calculate the value of α.

Given that all Link Nodes are derived from Data Nodes and all nodes in the XDCM are unique, some ancestor nodes may be calculated using different descendants in different paths. For example, in Fig.4, the node B in $all - B - C - D$ may be calculated by $all - B - D$ also. To avoid repeated calculations, we employ a "BreakPoint" to indicate the point at which to stop the calculation. The procedure for selecting a BreakPoint in the basic implementation is as follows:

1. The BreakPoint represents a node in a path. If the BreakPoint is going to be calculated in the next round, the calculation should be stopped.
2. Only Link Nodes and their ancestor nodes need to keep a record of BreakPoints. All nodes in the path of Data Nodes will be calculated. So it is unnecessary to keep records of BreakPoints for Data Nodes and their ancestor nodes.
3. BreakPoints are identified in the generation phase and won't change in the following calculations. The ancestor nodes of each Link Node should inherit its BreakPoint.

4. It should be noted that BreakPoints are the places where new branches are added while building the XDCM cube tree. For instance, in Fig.4, all the branches except $all-A-B-C-D$ are branches that were added. As for $all-A-C-D$, the BreakPoints are the nodes at A, because $-C-D$ is the branch we added.

Fig.4 shows the abstract structure of XDCM in four dimensions. As described in section 2.2, the paths of all nodes start from *all* and end at D. Thus, the total number of Link Nodes is the combinations from A to C minus one (because we already have the longest path, $A-B-C$). For the N-dimensional data, the number of Additional Nodes for each Data Node is $C_{N-1}^{N-2}+C_{N-1}^{N-3}+...+C_{N-1}^{0}(N \geq 2)$.

In our implementation, we use binary numbers to describe the status of those combinations. In Fig.4, the number of Additional Nodes for each Data Node is $C_3^2+C_3^1+C_3^0=7$. So using three bits $(2^{(3-1)})$ can describe all cases. Each bit stands for a dimension between *all* to D. If a certain bit is set to 1, it indicates having the constraint on that dimension. To identify a BreakPoint for each Link Node, we need to check whether the prefix of the Link Node's path has already existed in the tree that we have currently built. we may enumerate all of the cases by reducing the binary number from the maximum to zero. We determine the BreakPoint by comparing the prefix of two adjacent cases. The maximum of the binary number should compare with the binary number that stands for the Data Node. In the previous example, the 110 should compare with 111, so the BreakPoint for 110 is at the second bit which represents the B dimension. If there is no common substring in the prefix (as with 100 and 011), the BreakPoint for 011 is the node *all*.

4 OPTIMIZATION AND PERFORMANCE

4.1 Configuration Optimization

We performed the experiments in a cluster with 35 slave nodes each containing a 3.10GHz processor, 1 GB of RAM, and 40 GB of local disk allocated to the HDFS. Each node is running Hadoop version 1.1.2 on Red Hat 9.0 and connected by Fast Ethernet to a commodity router.

An XML document contains more information than an unstructured document with equivalent content. In order to handle XML documents in a distributed system without loss of information we need to restructure the data and divide it into small units. Thus, the amount of data through the system is larger than the input data. The table in Fig.5 shows the size of the data and the number of records through the system in the first round.

	Dimension	Map input bytes	Map output bytes	Map output records	Reduce input records	Reduce input records(after combining)
1	3	1, 023, 266, 669	4, 973, 220, 180	126, 360, 000	126, 360, 000	35, 101, 440
2	3	5, 386, 519, 202	24, 703, 200, 180	622, 080, 000	622, 080, 000	180, 001, 136
3	6	1, 104, 659, 389	44, 683, 382, 133	1, 075, 164, 192	1, 075, 164, 192	51, 916, 510
4	6	5, 514, 648, 663	130, 268, 589, 390	2, 686, 965, 696	2, 686, 965, 696	250, 036, 747

Figure 5: comparing the data through the system in the first round

Each record in Hadoop represents a unit we made which is described as a node in section two and three. In the experiments, we use two types of XML files for the experiments, of three-dimensions and six-dimension respectively. The arity of each dimension is nine. For 1 GB XML file in three dimensions, using the basic algorithm, the Map output in the first job is about 4.6 GB. In other words, to build the XDCM directly, the size of data we actually need is approximately 4.6 GB. For 1 GB XML file in six dimensions, using the basic algorithm, the Map output in the first job reaches 41 GB. Since Mappers and Reducers are separated, these outputs are transferred through the network. The I/O operations may have a negative impact on the whole task.

Hadoop offers several ways to optimize the I/O. In the experiments, we mainly used compression and combination, and their effects were clear. Hadoop supports several compression formats like gzip and snappy. The compression method should be selected according to the type of the task to be performed. If the task's CPU occupancy rate is high, it is better to choose a simple compression algorithm. If a significant portion of the task is spent at I/O, it is better to select an algorithm with a high compression ratio. in our experiment, using the default compression algorithm offered by Hadoop to process 1 GB of XML data in three dimensions provides approximately an 8% improvement.

In our experiments, the combination process provided a demonstrable improvement. The task of a Combiner is similar to that of a Reducer. If the Combiner is used then the outputs from the Mapper are not immediately delivered to the Reducer. The Key-Value pairs are collected in lists, one list per Key. The Combiner will process each list like a Reducer and emit a new Key-Value pair which has already been merged. Then the new Key-Value pair will be delivered to the Reducer as if they were created by the original Map operation. Fig.6 shows the performance using these two strategies when dealing with the three-dimensional XML files.

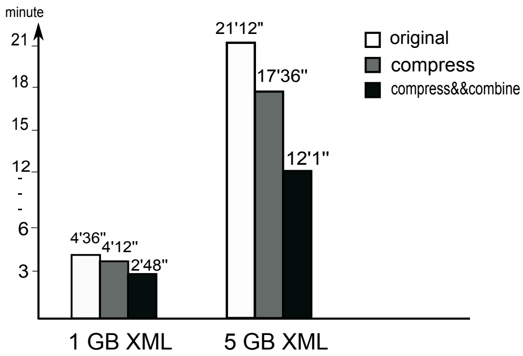


Figure 6: Performance of Optimization for 3-D XML.

4.2 Algorithm Optimization

For the 1 GB and three-dimensional XML file in our experiment, using the basic algorithm discussed in section 3, the number of Map output records is about 126,360,000 in the first round but the number of Reduce output records is approximately 35,101,440. About 72% records are merged in Reducers. For the 1 GB and six-dimensional XML file, the number of Map output records is approximately 1,075,164,192 but the number of Reduce output records is only 51,9156,510. The gap in the 5 GB file sizes is even more pronounced. As the dimension increases, the output of Map increases more rapidly. For a D-dimensional XML file, assuming n is the number of initial Data Nodes, the direct output records of Map is $2^D \times n$. In the basic algorithm, the workload is too large when the dimension becomes higher in the first round. Too much data is transferred from the Mapper to the Reducer. In a cloud environment with limited capacity, it may increase the risk of the jobs failing.

In the basic algorithm, all Additional Nodes come from Data Nodes. However, some nodes can be generated by Link Nodes also. The Link Nodes with longer paths can produce the Additional Nodes with shorter paths. In Fig.4, Nodes in $all-D$ can not only be generated by $all-A-B-C-D$ (Data Nodes), but also can be generated by other six types of Link Nodes like $all-A-B-D$, $all-B-D$, etc. Although the information in $all-A-B-D$ or $all-B-D$ is not as complete as the Data Nodes, it is enough for $all-D$. The significant advantage of using shorter-path nodes to generate Link Nodes is the decrease in the I/O between Mapper and Reducer. For example, there are only two Data Nodes, the paths of which are $all-a1-b1-c1-d1$ and $all-a2-b1-c1-d1$. Thus, the Link Node in $all-B-D$ will be $all-b1-d1$. It is more efficient to generate the Link Node in $all-D$ by using Link Node $all-b1-d1$ instead of using the Data Nodes because only one output record is produced by the Mapper, whereas the Data Nodes would produce two.

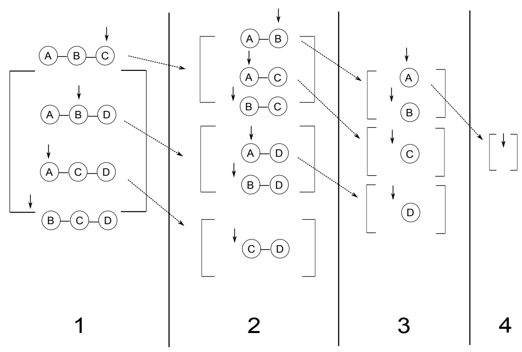
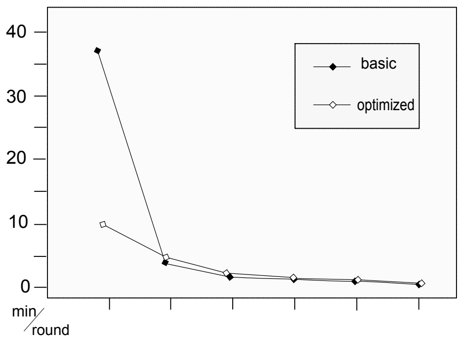


Figure 7: Using Link Nodes for Generation

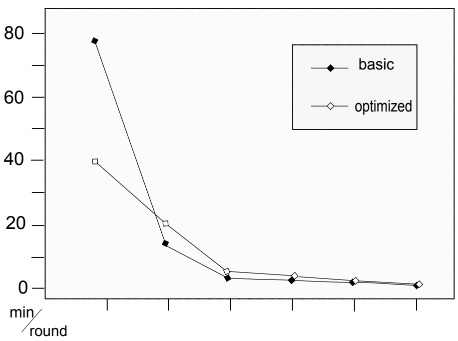
Fig.7 is an example of generating Link Nodes by using an optimized algorithm for five-dimensional data. As discussed in section 3, all of the possible Link Nodes for five-dimensional Data Nodes ($all-A-B-C-D-E$) depend on combinations of the path between all and E exclusively, which is $A-B-C-D$. In the first round, by removing one constraint in the path $A-B-C-D$, we get C_4^1 kinds of Link Nodes shown in column 1 in Fig.7. The arrow above the node is the indicator of a BreakPoint. The arrow pointing to empty means the BreakPoint is at all. Any Link Node whose BreakPoint does not point to all can be used to generate new Link Nodes in the next round. In the next round, another constraint of the Link Nodes is to be removed, and this constraint should be the nodes before the BreakPoint. As shown in Fig.7, the $A-B-C$ in column 1 stands for the Link Nodes whose path is $all-A-B-C-E$. The BreakPoint is at C, so in next round shown in column 2, successively removing the nodes before C (including C), which are C, B, A, we obtain $A-B$, $A-C$, $B-C$. The BreakPoint changes to the point at the node before the node has been removed. If the BreakPoint points at all like $B-C$, it would not participate in the generation work of future rounds, but the calculation phase would continue. In this way, we obtain all of the combinations after the removal of two constraints from the path, which is C_4^2 in total. The Link Nodes are generated according to the rules described above, until all BreakPoints point to all. The total number is $C_4^1 + C_4^2 + C_4^3 + C_4^4 = 15$, which is the same number produced by the basic algorithm.

Assuming the dimension is D, in the Nth round, the number of Additional nodes for each Link Node is C_{D-1}^N. It would appear that the I/O should increase when N is close to $\frac{D-1}{2}$. However, it does not because the Additional Nodes are produced by Link Nodes instead of Data Nodes, and after several rounds of mergers, the number of Link Nodes decrease significantly. The total outputs from the Mapper after N rounds are still less than the first round in our experiment.

Fig.8 shows the different efficiencies obtained by employing these two algorithms to construct the XDCM using the six-dimensional XML data. The improvement in the performance is significant. For the 1 GB and 6-dimensional XML file, the total improvement is approximately 64%. For the 5 GB and 6-dimensional XML file, the improvement is approximately 27%. Fig.9 shows the the performance of the two algorithms using the 5GB XML data from CityBikes.



1 GB XML in 6 dimensions



5 GB XML in 6 dimensions

Figure 8: Basic Algorithm vs. Optimized Algorithm

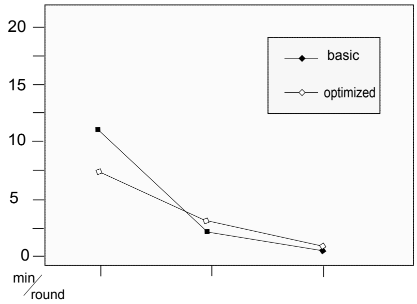


Figure 9: Performance in CityBike

5 CONCLUSIONS

In this paper, we illustrated a data cube model for XML documents to meet the increasing demand for analyzing massive XML data in OLAP. Hadoop is a popular framework aiming at tackling large-scale data problems. We proposed a basic algorithm to construct the XDCM on Hadoop. To improve efficiency, we offered some strategies and described an optimized algorithm. The result proves the optimized algorithm is suitable for this type of data and further enhance the efficiency.

REFERENCES

Dede, E., Fadika, Z., Gupta, C., and Govindaraju, M. (2011). Scalable and distributed processing of scientific xml data. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 121–128. IEEE Computer Society.

Dutta, H., Kamil, A., Pooleery, M., Sethumadhavan, S., and Demme, J. (2011). Distributed storage of large-scale multidimensional electroencephalogram data using hadoop and hbase. In *Grid and Cloud Database Management*, pages 331–347. Springer.

Gui, H. and Roantree, M. (2012). A data cube model for analysis of high volumes of ambient data. *Procedia Computer Science*, 10:94–101.

Gui, H. and Roantree, M. (2013). Using a pipeline approach to build data cube for large xml data streams. In *Database Systems for Advanced Applications*, pages 59–73. Springer.

Khatchadourian, S., Consens, M. P., and Siméon, J. (2011). Having a chuql at xml on the cloud. In *AMW*.

Lin, J. and Schatz, M. (2010). Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, pages 78–85. ACM.

Rusu, L. I., Rahayu, W., and Taniar, D. (2009). Partitioning methods for multi-version xml data warehouses. *Distributed and Parallel Databases*, 25(1-2):47–69.